CS247 Project Final Design

Overview:

This project is based around the MVC design pattern, with an emphasis on accommodating change. Our design separates the game logic (Model), user interface (View), and input handling (Controller), allowing for strong code maintainability and reusability. This also promotes low coupling with high cohesion, which follows software engineering principles and best practices.

Model:

In charge of the game's logic, rules, and states. The key components of the Model are:

GameState: This acts as the primary point of connection with the Controller to make changes. It owns the Board and list of Players, while containing high-level game-flow methods like *moveLink* and *downloadLink*, ensuring that game rules are applied consistently.

Board and **Cell:** These represent the physical game grid. The board is in charge of managing the grid of Cells. The Cell's *placeOccupant* method is responsible for automatically detecting and resolving interactions between links or abilities like battles and trap activations whenever a link is moved.

Occupant: Occupant is an abstract base class for anything that can occupy a Cell. This includes:

- **Link:** Concrete class that inherits from Occupant. Represents the links in RAIInet that are the pieces of a player. Each link has its own state, strength, type, and movement capabilities
- **Trigger:** Concrete class that inherits from Occupant. Used to represent all cell-based effects, ranging from abilities like Firewalls to battles between two links. Its responsibility is to wait on a cell until an event occurs like a link moving onto its position. Upon activation, it checks whether to execute based on the given change, and executes a specific piece of logic that is based on the type of trigger it is. A trigger can be used to implement core game rules, and custom abilities
    - Core rules, like how battles are resolved and Server/Download ports, are implemented by creating concrete subclasses (e.g., BattleTrigger, EdgeTrigger(download), ServerTrigger) that inherit from Trigger and contain this persistent logic. This handles all event-based interactions in the game.
    - Custom abilities create a lambda in their own file and pass it into a Trigger to sit on a cell.

Ability: Abstract base class for all abilities. It follows the Command Pattern with a pure virtual *execute* method. Each specific ability is a concrete subclass that has its own implementation of *execute* in order to support a wide array of abilities. This decouples the ability invoker from the ability's specific logic.

View:

In charge of displaying the game to the user.

View: Abstract base class that defines the interface for all display types and implements a caching system for lazy renders. The *printGame* and *printAbilities* methods determine how the game state should be rendered. This implements all of the notify handling and keeps a visual cache. It has a different function to tell displays exactly what has changed. This makes it so that the Controller can manage any type of View without needing to know all its details, which shows low coupling.

TextDisplay: Concrete subclass of View in charge of the text-based display in the console.

GraphicsDisplay: Concrete subclass of View in charge of the graphical display via X11. Uses View's difference function to do lazy rendering

XWindow: Utility class used by GraphicsDisplay which encapsulates the basic drawing and display operations for the graphical display. Demonstrates high cohesion since the low-level graphics code is contained within this one class.

Controller:

Converts user input into actionable information for the Model.

Controller: In charge of the application flow. Reads user input and parses it to determine what action should follow.

Command: Used to separate the process of parsing input from executing action. It uses the Command Pattern to create a command object (AbilityCommand, MoveCommand) and execute it. This is resilient to change since adding a new command type would mostly extend the program

- **AbilityCommand:** Used to handle a Player's request to use an ability. It takes the ability's identifier and its necessary arguments from the Controller, which are passed inside a Payload object. The *execute* method finds the correct Ability object on the current player by matching the identifier. Once the ability is found, AbilityCommand calls that ability's own execute method, passing only the relevant arguments in a new Payload.
- **MoveCommand:** Used to handle a Player's request to move a link. It takes the link's identifier and the direction from the Controller, which are passed inside a Payload object. The execute method parses the information from the Payload to identify the specific Link object and the intended direction. Once the arguments are resolved, MoveCommand calls the centralized *moveLink* method from Game State to perform the action.

Miscellaneous:

Payload: Utility class that allows us to transfer data. This is used almost everywhere to communicate between different modules, resulting in extremely low coupling as function calls are with a Payload.

Permission: A utility class that is used for managing ownership and visibility for all players, allowing the View to accurately depict the current game and allowing the Model to enforce rules based on the current player. Rather than having each Occupant and Ability manage its own visibility rules, each one contains a Permission object. This will allow us to very easily implement a 4-player Raiinet in the future!

MessageQueue: A singleton message broker that separates our communication logic between the Model and View. The Model posts GameEvent objects to the queue whenever a view change occurs. The View, as an observer, reads from this queue to know when it needs to redraw itself, without being directly called by the Model. This allows us to not store any high level class pointers or references in low level-classes, resulting in very low coupling.

GameEvent: Data structure that represents a specific change in the game state. It is created by the Model and placed on the MessageQueue to signal to observers that an update is necessary, containing information about what kind of update occurred.

Design:

We made sure to base our design around the idea of accommodating change and aiming for low coupling/high cohesion. From Due Date 2, some of the biggest design changes we made were:

1) Using weak pointers to prevent circular dependencies with Players and Permissions.
2) Changing getters/setters across the code to remove redundant ones and add ones that were required for any new features (see below)
3) Refactoring our notification process to use a MessageQueue instead of many specific notify functions. Initially, the Model needed to call specific notify's in the View. However, we updated this to accommodate change better by creating a GameEvent and placing it on the queue. The View classes monitor the queue and redraw when it sees an event, following the pub-sub model.
4) Changing our View parent class to act as a caching system to store all the necessary information for the subclasses (TextDisplay, GraphicsDisplay) to make changes and display the game. This was better than our initial plan of having each View childclass reimplement the notify function for all the different GameEvents that are possible and it also makes it easier for us to create newer views in the future
5) Introducing the Payload class to act as a data container, allowing us to pass parameters flexibly for various use cases. We pass a single Payload object which the receiving component is responsible for parsing. For example, when executing an ability, the Controller sends all user-provided arguments into a Payload and the concrete Ability subclass extracts the data it needs
6) Extending the triggers into subclasses rather than passing lambda functions as we do for abilities. This way the Cell isn't the one creating extra code that should belong in the Trigger. This just shows the extensibility of the pattern.
7) Creating the init function in GameState. Initially, we planned to create everything through the constructors directly, but because of the way different objects interact and have to wait for each other, we created the init function in GameState. For example we create all the links through the

players, but then to place them onto the board in their appropriate spots, we do this through the init function. This was necessary because otherwise we would have to give Player access to board to place them but that wouldn't make sense.

MVC Pattern

The entire project was built around the MVC framework. Doing so fulfils the Single Responsibility Principle. For example, replacing the text-based display with the graphic display on X11 would only require calling the GraphicDisplay class from the Controller instead of the TextDisplay class. This division also creates highly cohesive components and enforces low coupling by minimizing the dependencies between the core logic and the user interface.

Command and Factory Method Patterns

The Controller's communication with the Model was an example of the Command Pattern, facilitated by a Factory Method. The parseInput method in the Controller acts as a Factory Method. It takes user input and is responsible for creating a family of related concrete Command subclasses (MoveCommand, AbilityCommand). This encapsulates the object creation logic in one place. The AbilityCommand class then sends the request to a concrete ability, like Download or Scan. This entire mechanism encapsulates each action into a single, cohesive unit and cleanly prevents coupling the invoker with the actual concrete logic/operation, as the main control loop only needs to know how to execute a generic command. This adheres to the Open/Closed Principle. To add a new type of action (e.g., a ForfeitCommand), we can simply make a new subclass and extend the factory method. This would also make something like undo and redo, incredibly easy to implement.

Observer Pattern

The communication from the Model back to the View is handled by the Observer Pattern, a direct application of the Dependency Inversion Principle. For low-level modules within the Model (eg. Cell) to notify the view of a change in its Occupants, it must call upon either View or Controller's notify function, meaning, it must store, or get a reference to these high-level modules. Instead, a singleton MessageQueue was created to facilitate communication between low-level Model modules and the Controller. A low-level module can get a shared_ptr to MessageQueue and enqueue a message. The MessageQueue would notify all of its subscribers of the GameEvent (a subscriber being Controller). This design achieves perfect coupling (none) between the game logic and its presentation. The proof behind this is that we could attach a new logging View for a spectator mode without ever modifying the GameState Model.

Liskov Substitution Principle

The project's flexibility is built by closely following Liskov's Substitution Principle. Within the Occupant class, the Board is designed to manage a collection of abstract Occupant pointers. It can hold Links, Triggers, and BattleTriggers interchangeably because they all substitute for their Occupant base class, as advised in LSP. This allows the *placeOccupant* method, which is used in various places, to interact with any type of game piece This design is also open for extension as a new type of Occupant can be added

without any modification to the Board class. A practical application of RTTI we used was dynamic_pointer_cast to handle interactions between different Occupant subtypes. This allowed us to see when a Link is interacting with a Trigger or another Link.

Template Pattern

The template pattern is used for Triggers. The Trigger class defines the algorithm for a game-board object, which executes a specific action when activated via the *trigger* method. However, instead of requiring new subclasses of Trigger to define different trap behaviors, the variable part of the algorithm is injected at creation time via a lambda function. When an ability like Firewall creates a Trigger, it packages its triggerlogic and passes it to the Trigger's constructor. This approach avoids rigid inheritance and the need to create excess subclasses which can accumulate very quickly for a game meant to scale. To define a new ability, one can implement it entirely in a new file, with access to GameState and the option to create any trigger without creating a new class. This perfectly follows the Open/Closed Principle

RAII and Smart Pointers

A key design goal we aimed to meet was to complete the project without any manual memory management. This was accomplished by following the RAII idiom through the use of smart pointers. Every dynamically allocated object that we needed was immediately placed in a smart pointer, which ensured that its memory was automatically cleaned up when it went out of scope. We ran into a circular shared_ptr dependency error because our Players owned Abilities, which have Permission, which have shared_ptrs to Player. We fixed this using a weak_ptr in Permission to break this cycle.

Resilience to Change:

The most significant deliberation in our project was accommodating change. We knew that as we got deeper into stages of development, we would come up with new ideas and if we did not spend time building resilience to change, it would be costly in the long run. We wanted the introduction of new features and modification of existing ones to be minimally disruptive, which was done in two ways.

1) Following the principles and design patterns from class and
2) Developing a strong framework of classes to tackle both existing issues and issues we foresee from extending the game further

MessageQueue:

The MessageQueue uses the Observer pattern to create 1-way communication from the Model to the View, which is important for withstanding change. The Model publishes GameEvents without any knowledge of who is listening, meaning new listeners can be added to the system at any time with zero changes to the Model. For example, a ReplaySystem could be built to subscribe to the MessageQueue. Later, a PlaybackSystem could read this file to recreate the game step-by-step. This entire major feature could be implemented without altering any of the existing Model code.

Permissions:

The Permission class makes the system better at handling change. Instead of all of the Occupants and Ability objects implementing their own visibility and ownership logic, this is delegated to a Permission object. This means that if we decided to change the visibility rules (e.g team-based game mode or 4-player RAIInet), the changes would be almost entirely within the Permission class.

Triggers:

The Trigger class separates generic behaviour from specific behaviour, so a trigger can execute any action it is given without needing to know what that action is. We used this for abilities and game logic. There is no game logic (battles, downloads) on our board or game state, we have triggers placed around the board that do this for relevant cells. We extended our board from an 8x8 to a 10x10, containing a ring of cells all with download triggers on the top and bottom, and invalid cells on the left and right. Any change that requires any form of logic to be placed on the game board (whether ability or actual game logic), can use these triggers to easily implement. For example, if we want to create a new type of cell-based ability like a Freeze, we do not need to create a new subclass of Trigger. Or, if we wanted to create a 4-player game, we could just create a larger board, and scatter abilities around.

GameEvents/MessageQueue:

If any changes to game logic need to notify the views, they can pass a GameEvent through the MessageQueue. This message queue will notify the Controller, which notifies all of the views. Internally, the views contain a cache for minimal changes, and a get difference function to figure out what exactly to change. A change to game logic can use the message queue and use the GameEvent object to transmit the change directly to view, which automatically deals with the lazy refresh. If one must create a new notify function, they can choose to do so, only in the view class, and be sure that all their types of views will be able to use it.

Payload:

Payloads are used for dynamic parameters throughout the program. If you want to pass a reference to a specific object in the game for a function to use, we can simply pass an identifier to the object in the Payload instead of modifying the GameState to get the objects you need. This allows you to create almost anything that can interact with the existing program, just by sending and receiving custom Payloads, and defining your own interface for what the Payload should expect in each scenario.

Occupant:

The Board's complex logic for managing grid positions and resolving occupant placements interacts only with the abstract Occupant interface. Consequently, the system can be extended with entirely new types of game pieces with zero impact on the Board class. For example, a Capture the Flag game mode can include a flag occupant, which rewards points for capture. As long as Flag inherits from Occupant, the Board can store it, manage it, and place it on a Cell without any changes.

Command:

The Command Pattern ensures that the system is built to handle new user actions. The Controller receives a generic Command object from its factory method and calls the *execute* method, without

knowing what the command does. This means adding a new command is purely extension, not modification. The application's main loop remains untouched, isolating the change and minimizing the risk of issues arising from any changes.

Answers to Questions:

In this project, we ask you to implement a single display that maintains the same point of view as turns switch between player 1 and player 2. How would you change your code to instead have two displays, one of which is from player 1's point of view, and the other of which is player 2's point of view?

Our final implementation implements this actual feature for extra credit. It has a GraphicsDisplay class that manages two separate XWindow instances, providing a dedicated point of view for each player, with the correct permissions and visibility for each view.

This is slightly different from our answer on DD2. Instead of having the GameState (Model) manage multiple View objects, we encapsulated all window-management logic within the View class. This created a more cohesive design that better adheres to the Single Responsibility Principle by keeping presentation logic out of the Model.

How can you design a general framework that makes adding abilities easy? To demonstrate your thought process for this question, you are required to add in three new abilities to your final game. One of these may be similar to one of the already-present abilities, and the other two should introduce some novel, challenging mechanic. You are required to actually implement these.

Our framework makes adding abilities easy by combining an abstract Ability base class with the Command Pattern. Most new abilities can be created in just 1 file as the ability will have access to GameState, implement triggers, and send notifications to View. The Controller parses user input into an AbilityCommand, which finds and executes the correct ability by matching an identifier to the Ability's public name. Arguments are passed in a Payload object, minimizing coupling between the Controller and the needs of any ability. For persistent board effects, a Trigger class accepts a lambda function, allowing runtime injections. This framework allowed us to easily implement the three new abilities we proposed in DD2; swap links, barbed wire, and telescope.

This design aligns very closely with our DD2 proposal. The biggest difference was generalizing the arguments to the abilities via the Payload class, which was a better alternative to the list of strings approach from DD2.

One could conceivably extend the game of RAINet to be a four-player game by making the board a "plus" shape (formed by the union of two 10×8 rectangles) and allowing links to escape off the edge belonging to the opponent directly adjacent to them. Upon being eliminated by downloading four viruses, all links and firewalls controlled by that player would be removed, and their server ports would become normal squares. What changes could you make in your code to handle this change to the game? Would it be possible to have this mode in addition to the two-player mode with minimal additional work?

Our final structure supports a four-player game mode because it was designed to be extendible for more than two players with few changes. The adjusted board (+) can be created during setup by utilizing the obstacle cell type on a larger grid to block out the corners. The game's turn-management loop already cycles through a vector of players, making it scalable. Additionally, we treat the player's edge (where you can walk off to be downloaded) as an extra row of cells containing Download triggers for the game logic, so to accommodate 4 players this would simply be a matter of adding two columns on the sides for this same purpose, which is easily extendable. Therefore, supporting four players would primarily require changes to the initial game setup and extending the I/O layers to handle four sets of inputs and displays, rather than modifying the game rules, which supports our goal of accommodating change and scalability. This approach is also fairly similar to our DD2 plan. The successful implementation of features like Obstacle cells and a minimally coupled Model demonstrates that our architecture was built with this kind of scalability in mind.

Extra Credit Features:

We implemented several extra credit features that demonstrated our creative and critical thinking to approach challenges that improve not only the user experience of the game, but also the code quality.

Automatic Memory Management

As mentioned earlier, we implemented the suggested extra credit feature of fully non-manual memory management (no use of deletes). We based our approach from the RAII idiom taught in class. This design delegates all lifetime management to the smart pointers themselves, which automatically handle deallocation through reference counting. The biggest challenge was maintaining this consistently across the entire project, ensuring no raw pointers were used for ownership. In both Due Date 2 and Due Date 3, this process involved a lot of deliberation and foresight, since we had to consider the bigger picture regarding scope and intelligently using smart pointers. However, by implementing this feature, we can be confident in our memory management throughout the game, even if new changes are added to the current version.

Board Obstacles

Obstacles in the game board are cells that are blocked off to Players. Players must navigate around these cells throughout the game. We attached the randomization logic (used for Links) to these obstacles, which placed 3 random obstacles on the board. The main challenge was the added logic needed for extending our game to include this new feature. However, since we set up our code to follow the Open/Closed Principle from the start, this process was made much simpler. Instead of needing to modify our existing Board class, we were able to simply build off it to add a new Cell type. A naive approach would require every function related to movement or ability targeting to contain special logic to check for obstacles, leading to high coupling and duplicated code. This decoupled the game rules from the obstacle feature itself and also provided a solution for future extensibility. This is because custom board layouts like a four player map (+) can be created by simply adding obstacles to the corners, with minimal changes needed to the game's move or ability logic.

Board Overflow

Board overflow is a special feature of links to wrap-around from the edge of a board and appear again from the other edge (for sides not ends). The main challenge with implementing this feature was enforcing the existing rules and adding to them instead of needing to modify them for a "special case situation". If we just teleported a link by changing its coordinates, it would fail to trigger cell-based events (e.g Firewall ability) or initiate battles that might be present on the new cell. Our solution was to treat cell wrapping as a move operation. The destination of the cell would be calculated and then we invoke the same *placeOccupant* method that is used for every regular move. This design choice enforces all standard interactions without duplicating the rules or creating special-cased code which can raise maintainability issues in the future.

Preventing Peeks

To create a true two-player experience, we implemented a graphical display that provides each player with their own dedicated window. When the game starts in graphical mode, it opens two separate X11 windows, one representing what Player 1 can see and the other for Player 2. Each player's window only reveals information that they are permitted to see like their own link strengths, their revealed abilities, and any opponent information they have uncovered through gameplay. This prevents screen-peeking for competitive players and allows for a more strategic and fair game, as players can physically separate their views onto different monitors. The technical challenge involved is ensuring that updates are rendered smoothly and without visual tearing or race conditions. We addressed this by incorporating sleep calls within the rendering loop. This not only synchronizes the updates but also improves the overall user experience by creating deliberate pauses between actions, making the flow of the game easier to follow. Instead of a constantly flickering screen, each player sees a stable representation of their game state, which updates cleanly after each turn.

Final Questions:

1. *What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?*

The most important lesson this project taught us about developing software in teams was the importance of thorough planning early on. We were fortunate to emphasize our planning stages during DD2, where we designed the UML diagram and made the plan document while accounting for many different requirements of the problem. It was crucial that we set ourselves up for success from the start, and so we had many detailed discussions about every part of the architecture before we began implementing. Had we not put as much effort into DD2, we would have struggled significantly more during DD3 and spent a lot of time writing and rewriting code. Therefore, when working with a team on a software from scratch, it's crucial to gather everyone's opinions on the architectural details in order to get as many perspectives before sticking to one idea. Otherwise, later on you may realize that you overlooked a key detail which has many cascading effects. Apart from this, we learnt the value of maintaining a strong git workflow to allow for parallel work and referring to previous code versions in the case of an issue or conflict. This project illustrated to us that developing software in teams is less about

writing perfect individual code and more about defining and respecting the architecture we set up as a group.

2. *What would you have done differently if you had the chance to start over?*

If we had the chance to start over, the biggest thing we would do differently is to emphasize the very fine details of the game as much as we did for the high-level architectural planning. We were very focused on code extensibility, accommodating changes, and high cohesion/low coupling, but not as much on ensuring that we were all on the same page regarding very specific and low-level details like “is a firewall activated when a player steps on it, or after they move over it?” for example. Little miscommunications like this led to confusion later on since it affected specific implementation details but not the high-level architecture, so it never came up earlier. By clarifying all rules that were even slightly ambiguous early on, we could have prevented any development and knowledge discrepancies, ensuring that every team member is building from the exact same set of assumptions.